A Framework for Network Protocol Software

Hermann Hüni
GLUE Software Engineering
Ralligweg 10
CH-3012 Bern, Switzerland
fax: +41 31 301 4433
phone: +41 31 305 0311
hueni@glue.ch

Ralph Johnson
Dept. of Computer Science
University of Illinois
1304 W. Springfield Ave.
Urbana IL 61801
phone: 217-244-0093
johnson@cs.uiuc.edu

Robert Engel
Ascom Tech AG
Morgenstrasse 129
CH-3018 Bern
fax: +41 31 999 27 41
phone: +41 31 999 42 73
engel@tech.ascom.ch

Abstract

Writing software to control networks is important and difficult. It must be efficient, reliable, and flexible. Conduits+ is a framework for network software that has been used to implement the signalling system of a multi-protocol ATM[1] access switch. An earlier version was used to implement TCP/IP. It reduces the complexity of network software, makes it easier to extend or modify network protocols, and is sufficiently efficient. Conduits+ shows the power of a componentized object-oriented framework and of common object-oriented design patterns.

1 Introduction

A protocol stack is usually implemented with a layered software architecture. For example, TCP sits on top of IP, which sits on top of the Ethernet driver (See figure 1). Each layer communicates with the layers above it and below it. Requests move up and down the stack from users to the network and back to users again. It is not hard to build individual layers so that one can be replaced by another, such as replacing IP with SLIP[2]. However, a layered architecture does not address reusability across layers. Each layer requires a new implementation even though the different layers have much in common.

[1]Asynchronous Transfer Mode

Copyright © 1995 by the Association for Computing Machinery, Inc. Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that new copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request Permissions from Publications Dept, ACM Inc., Fax +1 (212) 869-0481, or permissions@acm.org.

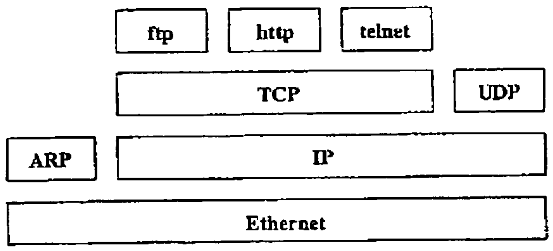

Figure 1: A TCP/IP Stack.

A key problem in designing reusable communication protocol software is how to factor out the common structure and behavior from the protocol specific parts. Solving this problem results in software components that can be reused with no source-code modification within a variety of different protocol implementations. Our paper shows how to achieve this by applying a number of previously described object-oriented design patterns. Two additional benefits are:

- The protocol specific software parts become simpler and easier to maintain and evolve since the framework provides the *infrastructure* operations.

[2]Serial Line Internet Protocol

- The result is a framework for network protocol software, in other words, a generalized software architecture for communication protocols. Documenting this framework helps communicate the underlying model and how it works.

Conduits+ is an example of a black-box framework, i.e. a framework in which components are reused mostly by composing instances (Johnson and Foote, 1988). In contrast, a white-box framework is one in which components are reused mostly by inheritance. White-box frameworks get their name from the fact that their users tend to have to know more about the implementation of the components they reuse.

Black box frameworks are usually easier to use than white-box frameworks, but are always harder to design. Most black-box frameworks started off as white-box frameworks and then gradually evolved to become more compositional. Conduits+ was designed this way, too. The paper describes how Conduits+ evolved from a white-box framework and so is a case study in how to design frameworks.

We describe the evolution of Conduits+ in terms of a sequence of design patterns. Design patterns are descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context. The design patterns we use have all been described elsewhere (Gamma et al., 1995), so we do not describe them in detail here. However, we describe the problem they solve and the design that results when they are applied. We have found patterns to be a useful way to describe the design of a framework (Beck and Johnson, 1994).

The graphic notation that we use to depict our architecture throughout this paper is similar to Booch's object-diagrams(Booch, 1994). This *object-scenario* notation was invented by GLUE Software Engineering primarily for teaching object-oriented software engineering (Hüni and Metz, 1992) using C++. While developing this framework, it was extensively used to communicate and document design-situations and has shown a lot of expressive power.

Conduits+ is being used in a commercial product. It shows that an evolutionary development of a framework is commercially viable, and that design patterns are useful in commercial projects.

2 Basic framework architecture

The framework is made up of two sorts of objects, *conduits* and *information chunks*. A conduit is a software component with two distinct sides, sideA and sideB. A conduit may be connected on each of its sides to other conduits, which are its *neighbor* conduits. A conduit accepts chunks of information from a neighbor conduit on one side and delivers them to a conduit on the *opposite* side. Conduits are bidirectional, so both its neighbors can send it information.

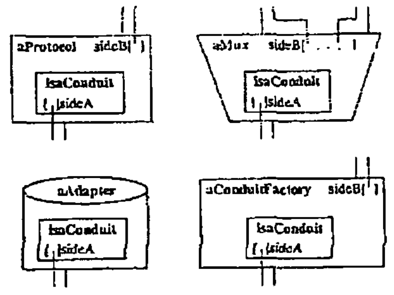

Figure 2: Four kinds of conduits.

There are four kinds of conduits as in figure 2, all of which have one neighbor on sideA. A Mux can have many neighbors on sideB, an Adapter has no neighbor conduit on sideB, and a Protocol and ConduitFactory have exactly one.

We will use the vague term *information chunk* for everything that flows through conduits. We will be more specific in Section 3 where we refine the basic framework architecture and make it more reusable.

Each layer of a protocol stack is implemented by one or more conduits. Thus, a TCP/IP stack would contain conduits for TCP and for IP, while an ATM signalling stack would contain conduits for the signalling ATM adaption layer (SAAL), the actual layer 3 signalling and the call control layer which is responsible for call routing. The information chunks in a TCP/IP stack include Ethernet, IP, and TCP packets. In addition, some of the information chunks represent operations like opening or closing a channel. Since an ATM switch transfers all data in hardware, all the information chunks in an ATM signalling stack represent operations.

Originally we only considered strictly vertically stacked layers of communication protocols and so called the sides of a conduit *upper* and *lower*. But conduits can also interconnect two stacks such that information chunks that travel upwards in one of them are moved to the top of a second stack to flow down towards the network. Such an inter-working[3] conduit is horizontal, proving that upper and lower sides do not always make sense. Also, some conduits may have an asymmetric structure or behavior. Our new terminology lets us reuse a given conduit upside-down without getting confused.

2.1 The Mux

A *mux* is a conduit that connects one sideA conduit to any number of sideB conduits. A mux multiplexes information chunks arriving on sideB to the single neighbor conduit connected on its sideA. Information chunks from sideA are demultiplexed to one of the neighbor conduits connected on sideB. The sideB instance variable of the mux in figure 2 denotes a *multi-valued* variable which might be realized trough an appropriate collection class like a List or a Map.

Each layer of a protocol stack can contain a mux. For example, in a TCP/IP stack, the Ethernet layer contains a mux to connect to low-level protocols like ARP and IP, the IP layer contains a mux to connect IP to UDP and TCP, and the TCP layer contains a mux to connect it to clients like ftp, telnet, or http.

A mux for a layer like Ethernet does not need to add new sideB conduits dynamically, because the low level protocols that use the Ethernet are usually specified when the operating systems kernel is built. But a mux for the TCP layer must be able to add new connections dynamically. Thus, it is important for it to be easy to install and disconnect sideB conduits.

[3]Mapping one type of communication protocol to a similar one.

A mux must be able to extract a dispatch address[4] from an information chunk arriving from sideA and demultiplex the information chunk to the neighbor conduits on sideB. For example, a mux in an IP layer must extract a service identifier from the IP message to tell whether to dispatch it to TCP or UDP, and an ATM signalling system must dispatch the information chunks to conduits representing individual calls based on a *call-reference*. In the opposite direction, a mux must provide an identifier that represents the sender sideB conduit so it can label the information chunk when it multiplexes it.

2.2 The Protocol

A communication protocol can be described by a finite state machine (FSM) and is implemented by a *protocol* conduit, which is where information chunks are produced, consumed, and tested. The protocol remembers the current state of the communication. It also provides commonly required facilities such as counters, timers and storage for information chunks to be temporarily retained. A protocol has exactly one neighbor conduit connected on both of its sides.

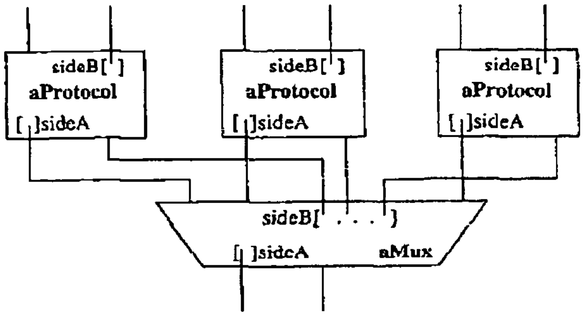

Figure 3: Three Protocols on a Mux.

In general, a single layer in a protocol stack will be implemented by several conduits, each specialized for a single role. For example, a TCP layer really consists of a protocol conduit to implement the protocol and a mux to dispatch packets to TCP connections. Thus, a layer in the protocol stack is a string of conduits.

[4]The terms address, identifier, index, dispatch-key and session-key all mean the same thing in different situations.

PAGE 15/26 * RCVD AT 12/14/2005 8:00:26 PM [Eastern Standard Time] * SVR:USPTO-EFXRF-6/24 * DNIS:2738300 * CSID:310 964 0842 * DURATION (mm-ss):10-34

2.3 The ConduitFactory

One important design problem is how to add new sideB conduits to a mux. Another design problem is what a mux should do with an information chunk from sideA that addresses a non-existing conduit on sideB. We solve both these problems by giving each mux a *default conduit* that is a neighbor conduit connected on sideB of the mux on the dedicated b0 channel. Each mux has exactly one default conduit, and zero or more other sideB conduits. In the IP layer, information chunks sent to the default conduit have illegal service identifiers, so we would use a default conduit that logs errors and responds to them. In ATM signalling systems, information chunks that are delivered to the default conduit indicate that a new call (or more commonly a new *session*) needs to be established, so the default conduit should install a new sideB conduit.

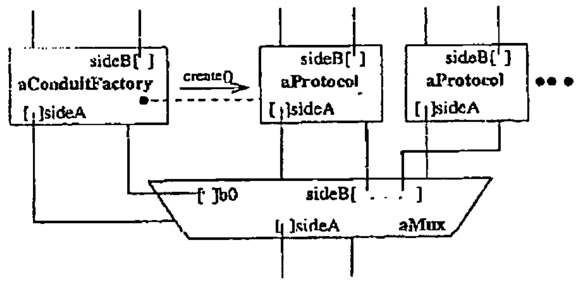

Figure 4: A Mux with a ConduitFactory as its default conduit.

Our architecture (See figure 4) can create and install new sideB conduits for a mux using a conduit factory as the default conduit to a mux. Whenever a new session has to be set up, the information chunk received from sideA of the mux will be delivered to its default conduit (on channel b0). It thus arrives at the conduit factory, which then installs a new instance of the required session protocol on sideB of the mux. That same information chunk that the mux was previously unable to handle will now revisit the mux and can correctly be demultiplexed to the just installed protocol conduit. The next time an information chunk with such a session key arrives at the mux, it will be directly handed off to that new session protocol.

The dashed line originating at aConduitFactory in figure 4 denotes a global reference[5] to the class of the object. The operation create() is applied, according to this reference, on class Protocol to produce the instance aProtocol.

2.4 The Adapter

An *adapter* is a kind of conduit that has no neighbor conduit on its sideB. Thus, only its sideA is connected to another conduit. The adapter conduit is used to interface the framework to some other software or hardware. Its sideB implementation is usually specific to a particular software or hardware environment. For example, an Ethernet conduit is an adapter for hardware, while a http-client conduit is an adapter for application programs wanting to be an http-client.

The adapter often converts information chunks to and from an external stream-oriented format. It might simply interface to some library written in C by registering a call-back function with the library and offering a sideB function that accepts a stream to read from.

2.5 Summary of Architecture Model

Conduits+ divides a network protocol implementation into the parts that process information, which are the conduits, and the parts that represent the information being processed, which are the information chunks. Conduits have two basic interfaces, one for connecting conduits to neighbors and accessing those neighbors, and one for handling information chunks. All conduits share these interfaces.

Although one might argue that separating function from data is not very object-oriented, Conduits+ uses object-oriented techniques heavily. First, it uses inheritance to organize a hierarchy of conduits. Class Conduit has four subclasses, Mux, Protocol, Adapter, and ConduitFactory, and each of them may also be subclassed. Second, it uses polymorphism to make it possible to connect any kind of conduit to any other. As long as each conduit only

[5] We use dashed lines for global relations as well as to depict classes or modules

assumes that its neighbors support the Conduit interface, any kind of conduit can connect to any other.

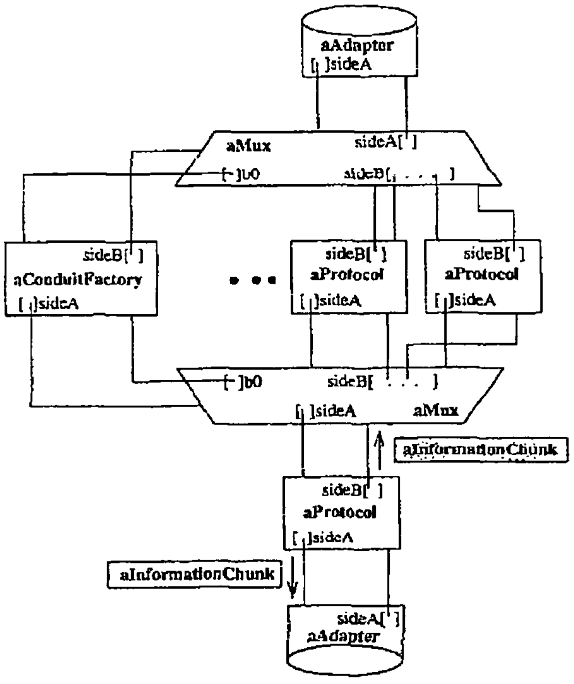

Figure 5: A typical conduit graph.

Since a conduit can be connected to any other kind of conduit, we can build arbitrary graphs out of these few simple components. Information chunks can then flow through the conduit graph to cause state-changes at protocol conduits. Figure 5 shows an example of a conduit graph as used to implement the layer 3 signalling protocol[6] in ATM (Engel, 1995).

3 Design Patterns improve the architecture

The architecture described in the previous section is not as reusable and extendible as it should be. In particular, a naive implementation of this architecture might require subclassing Protocol to define a new state machine, subclassing Mux to specify how addresses are extracted from information chunks, or subclassing ConduitFactories to describe the kind of conduit that should be added as a sideB conduit of a mux. These problems can all be solved using existing design patterns, resulting in a black-box framework that is elegant, highly reusable and easily extendible. We've already used some of these patterns, since Mux is an example of the Composite pattern and Adapter is an example of the Adapter pattern. But the next patterns that we use are more obviously *design patterns*, since they are being used only to provide a more flexible solution, not to model the problem domain better.

[6]Q.2931 as specified by ITU

3.1 Strategy Pattern

We want to be able to reuse the same Mux class at different places in the conduit lattice. Therefore, we need to configure a mux with a dispatch criteria. We call these components Accessors, since they access the relevant dispatch key within an information chunk. Moreover, a new conduit being installed on sideB of a mux sometimes needs a new session key according to some policy. This session key generator can also be hidden within the Accessor.

Each mux has an Accessor, which has two responsibilities. One is to compute the index of the sideB conduit to which the current InformationChunk should be dispatched. The other is to compute an index for a new sideB conduit. Thus, a mux will handle an information chunk that arrives on sideA by using its Accessor to get an index, and then sending the information chunk out on the sideB conduit with that index.

In figure 6, a reference to aInformationChunk is delivered as a method parameter from aConduit to aMux. The same reference is passed by method getDispKey to aTCPAccessor. To avoid cluttering this diagram with too many lines, we use the short-name ic of aInformationChunk to indicate this. When aTCPAccessor gets a reference to the information chunk, it invokes the provideTCPconnID method on it.

Separating an algorithm from the object that uses the algorithm is called the Strategy pattern. The

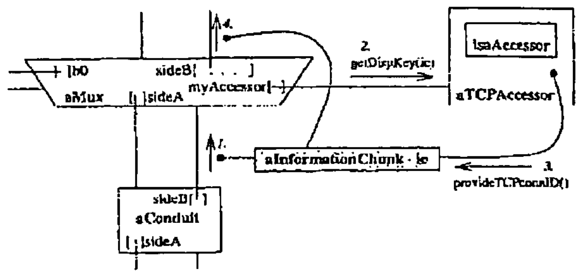

Figure 6: The strategy pattern object-scenario.

intent of the Strategy pattern is to let the algorithm, or strategy, vary independently from clients that use it. The mux is the *context* of the strategy, and the Accessor plays the role of the strategy. Accessors abstract out the difference between different mux objects on different layers in a protocol stack, so that the mux no longer has to be subclassed. Instead, a mux is given a reference to its Accessor when it is created.

The TCPAccessor knows how to extract a field from the TCP message that is its information chunk. In an ATM signalling system, the CallRefAccessor fetches the call number, while an EndPointReferenceAccessor gets the various parties in a point-to-multipoint situation.

3.2 State, Singleton and Command Patterns in concert

A communication protocol is usually realized by some (extended) finite state machine (FSM). It changes state when it receives InformationChunks or when a timer expires. Protocols are implemented using the State pattern, which means that each state of the Protocol is represented by a separate object. Protocols delegate their behavior to their state object, thus letting the protocol change its behavior when its state changes. A protocol changes its state by replacing its old state-object with a new one.

The State pattern in the Design Pattern book (Gamma et al., 1995) gives a broad interface to both the context (Protocol) and the state-object (State). This interface has an operation for each possible event. Thus, a TCP protocol conduit would have operations like OpenConnection, CloseConnection, and Send, and it would delegate all these operations to its state object. However, different protocols respond to different sets of events, so this would give different protocols different interfaces, limiting the reusability of Protocol.

Our version of the State pattern makes the protocol conduit more reusable. The protocol offers just one method, a method to accept information chunks. The information chunk interacts with the state object, usually invoking protocol-specific operations on it. Thus, State offers a relatively broad interface to the information chunks, but Protocol has a narrow interface. When a protocol is given an information chunk, it performs the apply(aState,aProtocol) operation on the information chunk, giving it both its state object and the protocol as arguments. The information chunk implements the apply operation by invoking an operation on the state object.

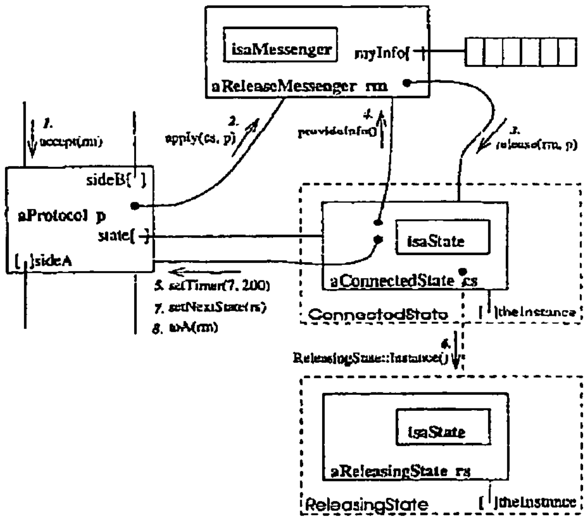

Figure 7: State, Singleton and Command pattern interactions.

Raw information chunks are often just arrays of bytes, so it is not appropriate to define operations like apply() for them. Instead, we introduce a new class Messenger that contains the information chunk and defines the apply() operation. A Messenger represents an event or an operation that is going to be invoked on the FSM of Protocols that it encounters as it is passed through the graph of conduits. Thus, it is an example of the Command pattern.

The purpose of the Command pattern is to encapsulate a request as a first-class object so it can be manipulated in some way. Using the Command pattern lets any conduit handle any Messenger by passing it along to a neighbor conduit. Protocol conduits handle Messengers by applying them on their current state object. This can not be done directly in a language like C++ where requests invoke methods and there is no first-class representation of requests to be passed around.

A communication system will probably have a large number of protocol objects in use at the same time, so the state objects should be sharable. This implies that they are immutable. This means that we have to store any mutable variables like timers, counters, etc. in the protocol object but manipulate them from the state object.

Each protocol requires a new State class hierarchy with a new derived class for each state in the finite state machine. Since there will always be at most one instance of such a state class, it makes sense to use the Singleton pattern for all state classes. Thus, there will be exactly one instance of each state class, and they will not have to be dynamically created or destroyed.

Figure 7 shows how a messenger interacts with a protocol and its state object. In this figure, the messenger represents the release operation. It arrives from sideB, is applied on the current state cs, sets timer number 7 to 200 msec., changes the state of the protocol from cs to rs, and then proceeds to the neighbor on sideA. Note that the definition of how the protocol responds to the release command in state cs is implemented only by its class ConnectedState. In general, all protocol specific code is in the State classes, with each Messenger just invoking a single operation on the state.

The Messenger and State classes are protocol specific. The state classes are usually not reusable from one communication application to another, though some of the Messenger classes might. Class State is tightly coupled to all of the Messenger classes, since it must implement all the operations that any of them perform on it.

Notice that messengers and states use double-dispatching (Ingalls, 1986). The first dispatch is to the messenger, and it performs the second dispatch on the state. Thus, the eventual function on the state depends on both the class of the messenger and the class of the state. This is exactly the situation that we expect for a transition in a FSM.

3.3 Visitor Pattern

A typical messenger is routed through a mux until it gets to a protocol, where it interacts with a state. However, some operations must traverse the conduit graph differently, especially when conduits are being added or removed from a mux. One alternative is to add new kinds of accept operations to Conduit, but that would defeat the purpose of the Command pattern. Another alternative is to make the messenger responsible for traversing the conduit graph, but that would couple Messenger to Conduit. Since the Messenger class hierarchy is already strongly coupled to the State classes (which are protocol specific), it is better to capture the non-specific and thus reusable algorithms for conduit-traversal in a separate class.

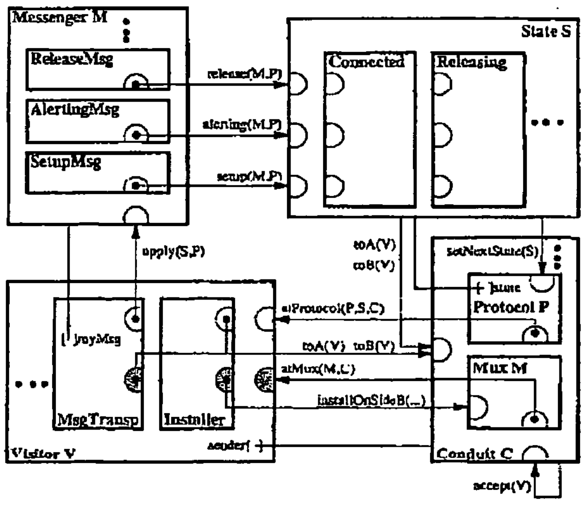

Figure 8: Framework class collaboration graph.

The solution to this problem is the Visitor pattern.

The intent of the Visitor pattern is to represent an operation to be performed on the elements of an object structure, which in this case is the conduit graph. Visitor lets you define a new operation on conduits without changing the classes of the conduits. It does this by making these operations on conduits be subclasses of the abstract class Visitor. Conduits will only have to deal with Visitors, so they will not depend on the Messenger hierarchy.

Visitor has several subclasses. One of them is MsgTransporter, which carries a Messenger around the conduit graph. Some Visitor subclasses help change the conduit graph. Installer is a kind of visitor that usually originates from a conduit factory. It installs a conduit object on the first Mux sideB it encounters. Figure 8 presents the framework's most important classes in the form of a class collaboration graph (Wirfs-Brock et al., 1990).

A visitor arrives at a conduit when the accept(V) operation is invoked by a previous conduit. The conduit then performs an operation on the visitor that indicates the class of the conduit. If it is a mux then it performs the atMux(M,C) operation, if it is a protocol then it performs the atProtocol(P,C) operation, and so on.

Thus, the visitor gets to know the conduit's subclass through the method that is invoked on it. Since the first argument is of type *pointer to the subclass* the visitor is able to invoke specific (subclass only) methods on this argument. The method installOnSideB(...) in figure 8 is such a method that is only available in the subclass Mux and is invoked from within the method Installer::atMux(M,C).

The visitor therefore decides on the appropriate action based on the conduit type, its own type and additional information it carries along.

When a MsgTransporter encounters a mux, it must traverse it, so it calls toA(V) or toB(V) depending on its direction. But when encountering a protocol as in figure 9, it should apply its messenger on the protocol's current state object. Other visitors may behave differently when encountering each kind of conduit.

While messengers are commands for states, visitors can be thought of as commands for conduits. But while messengers will always just execute a method

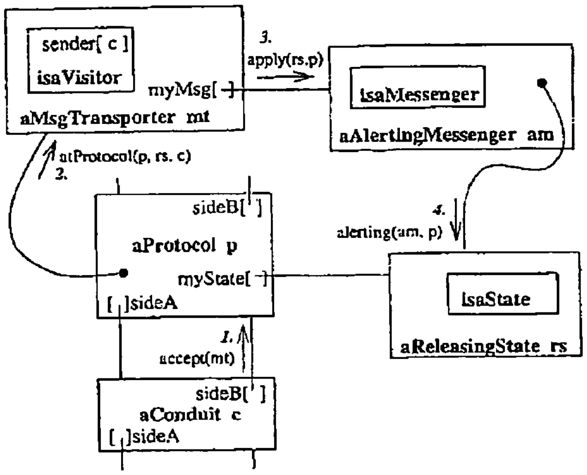

Figure 9: A MsgTransporter encounters a protocol on sideA.

of a given name on their target state object, visitors may do different things based on the *kind of conduit* they encounter. The effect of a messenger on a state (a transition) is determined by the state, but the effect of a visitor on a conduit can be decided by the visitor. This is a good trade-off because new states are more likely than new messengers, and new visitors are more likely than new conduits. The visitor pattern is like a more powerful command pattern because the visitor may initiate whatever is appropriate for the kind of the conduit it encounters.

Visitors must traverse conduits in both directions. One way to accomplish this is to give conduits two kinds of accept operations, acceptFromA() and acceptFromB(). Each kind would pass the visitor along to the other side. But this complicates how conduit graphs may be assembled because the previous conduit must know whether it is connected to a sideA or a sideB of its neighbor conduit.

Instead, conduits have only one kind of accept operation, and visitors are responsible for traversing conduits. One approach would be to have visitor subclasses VisitorFromAtoB and VisitorFromBtoA. But this would lead to too many subclasses and could not deal with two interconnected sideA's. The real notion of direction during this traversal is just *going to the opposite side* from where the visitor arrived. A visitor always knows its previous sender conduit, so it can figure out whether or not it arrived on the conduit's sideA and thus determine the opposite side. When conduits invoke an atXXX method upon a visitor, they always pass their sideA neighbor conduit as the last parameter.

The complexity added by the Visitor pattern is compensated by the fact that a Visitor can travel over a whole string of conduits before encountering its destination kind of conduit. In particular, it is easy to implement switching and routing functionality, as we will see in section 4.

3.4 Prototype Pattern

A conduit factory must be parameterized by the kind of conduits that it creates. An easy solution is for each conduit factory to define a function that creates its conduits (the Factory Method pattern), but this would require a new subclass of ConduitFactory for each new kind of conduit factory. A better solution is for each conduit factory to have a prototypical conduit that it copies when it needs to create a new one. This is an example of the Prototype pattern, which is specifying the kinds of object to create by using a prototypical instance. Using the Prototype pattern requires that conduits be able to copy themselves. It permits conduit factories to be specified by parameterizing an instance of ConduitFactory with a conduit graph that is an example of the conduits to be installed when a Mux needs a new sideB.

4 Call routing in an ATM signalling system

ATM is too fast to rely on software to switch data. The result is that ATM separates signalling information from data traffic, with the software handling the signalling information and the hardware switching the data. The ATM software steers the data by controlling the switch. It keeps track of the state of the calls and changes the switch when calls change state. Even though data is not flowing through the conduit graph, the conduit graph stores the state of the calls, so there will be a path of conduits for each call.

One of the most difficult problems in applying Conduits+ to ATM software is handling call routing. An ATM switch typically supports many trunks, and a call can originate in any trunk and end in any trunk. The trunks are connected by a cross-bar switch, but a Mux is a 1-to-N switch, not a N-to-N switch. However, Conduits+ can represent this situation, too.

The object scenario in figure 10 shows a simplified[7] version of the initial conduit graph for call handling and routing in an ATM signalling system. Only two trunks are shown, but there can be any number of them. The mux at the top is used together with a specialized visitor to establish new connections between trunks. Note that it is connected to the conduit factories of each trunk's mux.

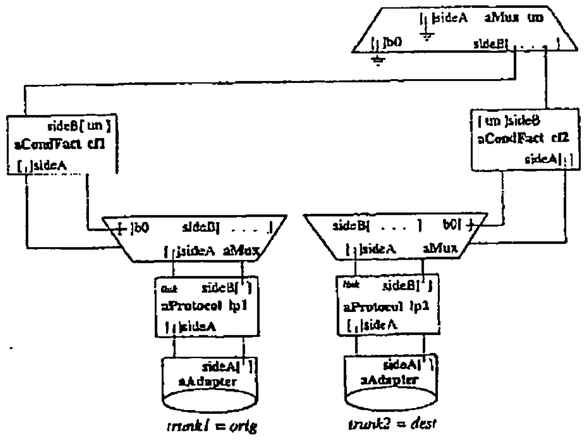

Figure 10: Initial conduit graph for 2 trunks with no active calls.

Initially, when no calls are in progress or established, every trunk in the system has one adapter, one link protocol (lp1, lp2), a mux and a conduit factory. Each new call will create a string of new conduits.

Suppose a new call is initiated on trunk1. The adapter creates a MsgTransporter with a SetupMessenger, which will arrive at the conduit factory cf1

[7] Ascom's multi-protocol ATM access switch is able to handle point-to-multipoint connections. This basically involves one more demultiplexing stage.

to create and install a new protocol sp1 (for layer 3 signalling) followed by a new protocol cp1 (for call-control). These two conduits are both instances of our single Protocol class. The protocol-specific behavior is caused by configuring the new protocols with suitable state objects. The conduit factory installs that string of conduits using two Installer objects (not shown here), each of which installs one side of the conduit string. SideA of the protocol sp1 is interconnected with sideB of the lower Mux on trunk1 while sideB of the cp1 protocol is connected to sideA of the top Mux tm. This intermediate result is visualized by the object-scenario of figure 11.

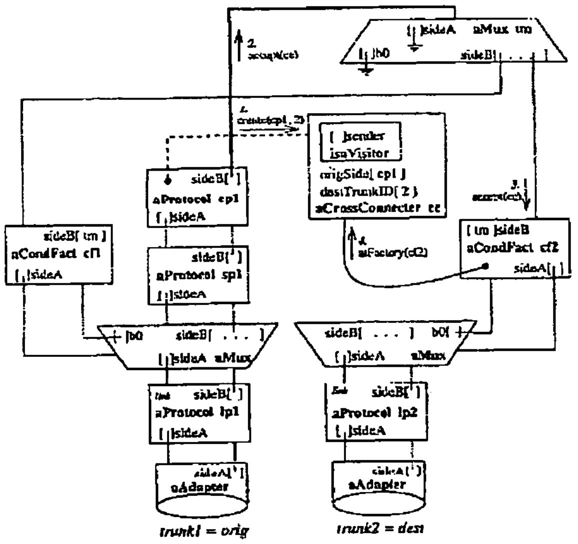

Figure 11: Routing a call with a CrossConnecter from one trunk to another.

The MsgTransporter with its SetupMessenger will now visit protocol sp1 and then protocol cp1. Here, the call has to be routed from its origin trunk to some destination trunk. Protocol cp1 is responsible to initiate this. It sets up a path to the destination trunk by creating a special visitor called a CrossConnecter, which remembers the cp1 protocol that created it and also a destination trunk-ID. The CrossConnecter cc is sent by the cp1 protocol on its sideB, immediately visiting the top mux tm, at which point it is dispatched, based on its destTrunkID, to the conduit factory cf2 of the correct destination trunk.

At this point, the CrossConnecter will ask the destination conduit factory to provide a new destination conduit string through the same mechanism as the origin one did. This leads to the creation of a new signalling protocol sp2 and a new call-control protocol cp2 as shown by the object-scenario of figure 12. The CrossConnecter will finally have to establish the interconnection of the two protocol conduits cp1 and cp2.

The SetupMessenger, which has been remembered by the origin cp1 protocol, can now continue to be carried by a MsgTransporter from sideB of this protocol to sideB of the destination cp2 protocol and eventually down the destination protocol stack to the adapter of the destination trunk.

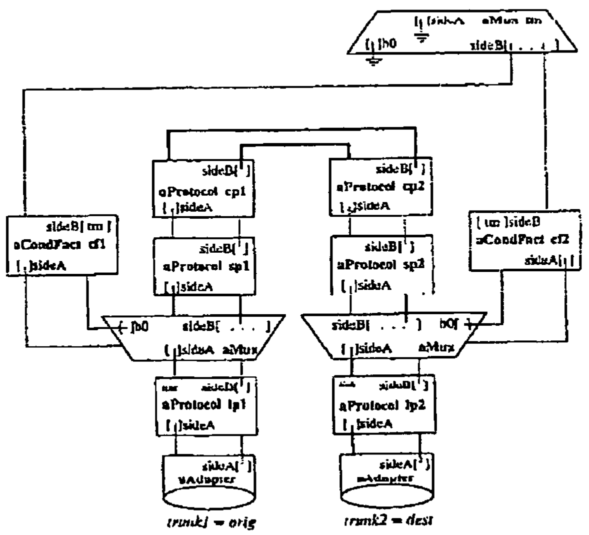

Figure 12: The final interconnected conduit path for the new call.

Other operations in an ATM signalling system are much more straightforward. For example, when a call is released, the entire conduit string between the origin mux and the destination mux has to be destroyed. A release request may be initiated from both parties connected by the call. Every protocol conduit will usually handle this event gracefully within its FSM. Before a conduit is destroyed, it has to inform all neighbor conduits on its sideA and sideB through an object of class Disconnecter, which is some kind of Visitor. The Disconnecter object must remove the reference from the neighbor conduit to the dying conduit.

5 Framework evolution and related work

The framework started out as a white-box framework that relied heavily on inheritance, and gradually became more black-box. Conduits was originally designed by Jonathan Zweig for his MS thesis (Zweig and Johnson, 1990) and used to implement TCP/IP. But this version did not separate Mux from Protocol, did not use the Visitor pattern, and did not have ConduitFactory. Thus, each kind of conduit required a separate subclass, and adding new protocols required much more development and testing.

Another excellent source of inspiration, although just using object-based technology, was some follow-on work of the x-kernel group on a language for protocol implementations called *Morpheus* (Abbott and Peterson, 1993). The authors claim that: *Morpheus' benefits could not be duplicated by adding predefined classes to a general object-oriented language like C++ since it would lack the knowledge of common patterns of protocol operation invocation that Morpheus exploits to optimize.*

While our framework is unlikely to achieve the same execution efficiency as a special-purpose programming language, it offers similar, but more easily extensible, compositional facilities. An important advantage is that it can be implemented with a standard C++ compiler.

In September 1993, a team at Ascom Tech started the design of ATM signalling software, using Conduits as a starting point. The team was small and only one member had extensive experience with object-oriented technology and C++. By January 1994, the framework had been used to implement the layer 3 ATM signalling protocol (Q.2931). Since then, the framework has undergone a number of major revisions and extensions, and is being used to implement multiple layers of the signalling system within a flexible multi-protocol ATM access switch being marketed today.

The first version of the ITU Q.2931 signalling protocol implementation was based on an *unapproved* specification. The *approved* version introduced mainly one new feature. The feature required the link protocol (lp1 in figure 12) to be able to issue a SynchronizationMessenger to all signalling protocol instances (like sp1 in figure 12).

At first glance this requires just a minor extension. However, with a classical approach, it is not that easy to broadcast such control information to all currently running protocol instances. Our framework solves the problem by adding a *BroadcastVisitor*. A BroadcastVisitor has a special behavior when it encounters a Mux on sideA. Instead of demultiplexing to some sideB conduit, it broadcasts to all conduits connected on sideB of the Mux. The effort to design, implement and test this extended feature was less than a week's work.

There are certainly more refinements to come. The currently hardest question is how factor the slight differences between network- and user-side protocols, between upwards compatible revisions of the same protocols, between protocol variations of different standard-bodies (ITU, ATM Forum) and between incremental feature sets to be added on demand to a protocol. We are currently investigating solutions that permit even higher levels of reuse in the area of States and Messengers.

Most new protocols require creating only new state classes. Conduits, Visitors, and even Messengers can usually be reused. But state classes have very regular and predictable interfaces, and it seems likely that they could be constructed automatically from higher-level specifications (ITU-T, 1993). Prototype based ConduitFactories are specified by graphs of conduits, and the initial protocol stack is a graph of conduits. It seems likely that both can be specified graphically. Thus, it seems possible to create a *protocol builder* in the same sense as there are many object-oriented interface builders.

6 Conclusion

Software for network protocols that meets international specs is often quite complex and hard to build. We have shown how to segregate a reusable infrastructure of black-box components into a cohesive framework and encapsulate the protocol specific parts into a few extra classes that are relatively easy to build and maintain.

Our design developed in steps. Particular reuse problems led to design patterns that solved them. The result was that most of the components of the framework became protocol independent. This means that someone using the framework can concentrate on just that part of the framework that needs to be customized, and can pay less attention to the rest. This leads to a framework that is much easier to learn and use.

This paper has concentrated on an architectural model and how design-patterns improve its flexibility and reuse-potential. This is only one of the issues that must be faced when designing a framework. Questions like memory management policies, scheduling of potential parallel activities, execution efficiency and testing facilities are also important but our solutions to these problems are not explained here.

Even though the design will continue to improve, it has already proven its worth. It has allowed a small group to develop complex software in a short amount of time, and has allowed us to rapidly change our software to track the proposed standards as they are revised.

We would like to thank Toni Bieri, Walter Bischofberger, Erich Gamma, Brian Foote, Philippe Oechslin and Lennart Ohlsson for their constructive comments on earlier drafts of this paper.

References

Abbott, M. B. and Peterson, L. L. (1993). A language-based approach to protocol implementation. *IEEE/ACM Transaction on Networking*, 1(1).

Beck, K. and Johnson, R. (1994). Patterns generate architectures. In *European Conference on Object-Oriented Programming*, pages 139–149, Bologna, Italy. Springer-Verlag.

Booch, G. (1994). *Object Oriented Analysis and Design with Applications*. Benjamin/Cummings.

Engel, R. (1995). Signalling in ATM networks: Experiences with an object-oriented solution. In *International Phoenix Conference on Computers and Communications*. IEEE.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns, Elements of Object-Oriented Software*. Addison-Wesley.

Hüni, H. and Metz, I. (1992). Teaching object-oriented software architecture by example: The games factory. In *OOPSLA Educators Symposium Proceedings*. ACM.

Ingalls, D. (1986). A simple technique for handling multiple polymorphism. *SIGPLAN Notices*, 21(11).

ITU-T (1988-1993). *Recommendation Z.100: Specification and Description Language*. ITU-T.

Johnson, R. E. and Foote, B. (1988). Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2):22–35.

Wirfs-Brock, R., Wilkerson, B., and Wiener, L. (1990). *Designing Object-Oriented Software*. Prentice Hall.

Zweig, J. M. and Johnson, R. E. (1990). The conduit: a communication abstraction in c++. In *USENIX C++ Conference*. USENIX.